# FrAid language basics

NOTICE: ""

## Table of contents

## 1. Atoms

A couple of somewhat simplified rules to give you an idea of FrAid's grammar:

- FrAid is case sensitive
- everything which begins with a letter is a function name (except for if, then, else, i, I, j, J, and the local function variables)
- everything which begins with a number yelds either a number or a lexical error (34rt is bad, 34.54e-17 is good)
- everything between a pair of double quotes "" is a string (unless preceded by '\')
- everything has a value, i.e. evaluates to something
- the assignment operator '=' is used only to (re)define functions
- the rvalue of '=' is always a single entity (a local variable or a function). In other words there is no sequencing within the user defined FrAid functions
- everything between curly brackets ('{' and '}') is a meta expression and a new hidden function will be generated and passed as an argument for it;
- curly brackets can not be nested
- the symbols in the table below constitute FrAid's operators and special symbols:

| | | | | |
|---|---|---|---|---|
| if - part of conditional expression | then - part of conditional expression | else - part of conditional expression | + - plus operator | - - minus operator |
| * - multiplication operator | / - division operator | ^ - power operator | ( ) - spec. priority or argument list | = - assignment operator |
| == - equal to | != - not equal to | > - greater than | < - less than | >= - greater or equal |
| <= - less or equal | & - logical and | \| - (vertical bar)logical or | "" - double quotes | ! - logical negation |
| # - logical xor | ; - end of FrAid line | - - unary minus | , - argument separator | // - single line comment |
| /* - comment start | */ - comment end | i, I, j, J - imaginary one | | |

## 2. Grammar

It goes roughly like this (in BNF):

- FrAidProgram ::= [Command;](Command;)*
- Command ::= FunctionDefinition | Assignment | Expression
- FunctionDefinition ::= Name '(' [Arguments] ')' = Expression

- Assignment ::= Name = Expression
- etc.

For details see the complete grammar

Both FunctionDefinitions and Assignments are interpreted to functions, so I don't describe them separately.

Every Command evaluates to something, so make sure to check the "**Evaluates to:**" field after each description bellow.

## 3. Semantics

### 3.1. Commands

- FrAid is interpreted. You send a command or a sequence of commands and the interpreter responds:

```
fraid:16>2+2;
4.0
```

- FrAid is case sensitive: `aaaa = 5;` is different than `aaAa = 5;`
- The commands can be single-line or multi-line (for better readability):

```
f(x)= if x == 1 then "one"   else
if x == 2 then "two"   else
if x == 3 then "three" else
"Error";
```

  reads much better than:

```
f(x)= if x == 1 then "one" else if x == 2
then "two" else if x == 3 then "three" else "Error";
```

- Every command ends with a semicolon, ';'.

### 3.2. Function Definitions

A fundamental idea in FrAid is for functions to look and feel as closely as possible to mathematical functions. Function declaration involves a function name, a list of arguments and a function body, which establishes some sort of mapping between the domain and the range of the function. Within the function body you can refer to:

- any argument from the argument list (you don't need to use all of them);
- global constants (Pi, E, etc.);
- other predefined functions;

- the function being defined itself (recursion).

Since all a function does in FrAid is map this element from the domain to that element from the range, you cannot have a sequence of commands withn a function definition (a bit like the C comma operator). Outside a function, of course, you can:

```
                    a=5; b=a+6; ...
```

The upside of this is that FrAid functions are very simple. The downside is that - as in every programming language - from time to time you need to perform a sequence of steps. Currently if your functions are not strictlly mathematical (where usualy this kind of sequencing is needed) you can use any of the existing logical or mathematical two argument operators to chain two or more commands (this will work because everything evaluates to a Complex):

```
                    f(x) = if isString( x )
                    then dump( ) & doSomethingWithThisString( x )
                    else "Error";        //here the "logical and" operator
is used to chain two function calls
```

Another solution is to develop a (Java implemented) Library function which takes a list of functions and calls them in sequence. This is not available at the moment. Alternatively, if all this proves to be too limiting, a sequencing comma operator may be provided in future revisions.

What is equivalent to variables in other languages is treated in FrAid as functions without arguments. The empty pair of parentheses after the function name is optional:

```
                    a=5;     //is equivalent to
                    a()=5;
```

A function is identified through its name only (i.e. the names are unique):

```
                    f(x)=sin(x)+cos(x);
                    f(x,y)=(x+y)^2;          // f(x,y) overrides f(x)
                    f(Pi);                   // error, f(x) is not
accessible any more
                    f(2,3);                  // --> 25
```

**Note:**

This changed from previous versions where a function was identified by its name and number of arguments (i.e. functions with the same names were allowed as long as they had different number of arguments.).

In a function definition the names of the variables don't matter

```
                    f(x,y)=x+y;    //is equivalent to
                    f(a,b)=a+b;
```

The body of the function may contain references to local (mentioned in the argument list of the function) or global (previuosly defined functions with no arguments), and if there are local and global variables with the same name, the local ones take precedence:

```
                    a=5;
                    f(x)=x+a;  // x is local, a is global
                    g(Pi)=Pi;  //Pi which is a global constant is not
accessible within
                    //g's body because there is a local variable with the
same name
```

**Evaluates to:** the assignments allways evaluate to zero:

```
                    a=5;         // global variable definition, evaluates
to 0
                    f(x)=x*x;    // function definition,        evaluates
to 0
```

## 3.3. Expressions

### 3.3.1. Numbers

There is only one numeric type in FrAid - Complex with double precision real and imaginary part. If you only work with real numbers you don't have to use the imaginary part. The imaginary one can be denoted by 'i', 'I', 'j' or 'J'.

All these are valid numbers:

```
                        2;
                        -2.45E18+.0098i;
                        -2.45E18-.0098e-34i;
                        1+i;
                        i;
```

**Evaluates to:** the number itself

### 3.3.2. Strings

A sequence of characters between a pair of double quotation marks - "xxx".

The following combinations of symbols have special interpretation within a string:

```
                        \n, \r, \\, \f, \b, \t, \"
```

Some valid FrAid strings:

```
                        "key";
                        "He said \"Hello.\"";     // note the '\' symbols
before the quotation marks inside the string
                        "Complex
                        Number";                 // this is the same as
next line:
                        "Complex\nNumber";
```

**Evaluates to:** (0+0i);

## 3.3.3. Function calls

Four kinds of arguments may be used when calling a function:

- Constants:
  - Strings: `load( "myfile" );`
  - Numbers: `abs( -5 );`
- Expressions which evaluate to something:
  - Strings: `isString( dump( ) );`
  - Numbers: `sum( sqrt( 5 ), abs( var1 ), var2 );`
- Functions passed as arguments: `plot( sin );` // the name of the function is sufficient

  The function called have to expect another function to be passed and know what to do with it, otherwise function references passed as arguments evaluate just to zero: `sin+cos; // --> 0`

**Evaluates to:** whatever the evaluation of the parse tree yields (see printTree().)

**Note:**

The postponed evaluation operator is deprecated in FrAid 1.5. Functions are passed as argument just by their name and the appostrophe symbol would generate a syntactic error.

## 3.3.4. Comments

You can place comments anywhere in your script.

The FrAid interpreter recognizes two kinds of comments:

- C-style: `/*this is a comment*/`
- C++-style: `//comment to the end of the line`

**Evaluates to:** comments are not evaluated by the interpreter.

### 3.3.5. Conditional statement

**if Expression1 then Expression2 else Expression3;**

**if this_function_doesnt_return_0( ) then do_something( ) else do_something_else( );**

Conditional statements can be nested and chained:

```
                        f(x)= if x == 1 then "one" else
                        if if x==3 then 1 else 0 then "two" else
                        if x == 3 then "three" else "Error";
```

**Evaluates to:** If Expression1 evaluates to a nonzero number Expression2 is evaluated and returned, otherwise Expression3 is evaluated and returned.

### 3.3.6. Meta expressions (closures)

Meta expressions are just short hand for function definitions which can be used where a function is expected to be passed as an argument. The result from the execution of these two scripts is the same:

```
                        //Example without meta:
                        f(x)=sin(x)+cos(x);
                        plot(f);

                        //Example with meta:
                        plot({sin+cos});  //notice how the functions are
passed to the meta without arguments
                        //only by names (just like being passed as
arguments to another function)
```

Meta expressions can make a program more readable. The negative side is that the semantics may be a bit confusing since a new hidden function is generated and passed instead of the expected argument.

**Evaluates to: a new hidden function.**

### 3.3.7. Recursion

The only way to repeat actions in FrAid is through recursion (if you need any of the for, while, do, etc. constructs you need to implement a FrAid function in Java). Here are some examples of recursive definitions implemented in FrAid:

```
                        sumR(x,y) = if y == 0 then x else sumR(x,y-1)+1;
//sum of positive numbers through increment
                        mulR(x,y) = if y == 1 then x else mulR(x,y-1)+x;
//prod. of positive numbers through sum
                        powR(x,y) = if y == 1 then x else powR(x,y-1)*x;
//power through multiplication
                        factR(x)  = if x == 1 then x else factR(x-1) *x;
//factoriel
```

And here is a bit more elaborate example how recursion can be used to animate your graphics:

```
                        a=1;
                        iterate( startValue, endValue, step ) =
                        if startValue > endValue
                        then 0
                        else   assign( a, startValue ) &
                        iterate( startValue + step, endValue, step );
                        f(x)=sin(a*x);
                        plot(f);
                        iterate( 1, 20, .2);
                        //now to run the animation again just call a=1;
iterate( 1, 20, .2);
```

## 3.4. Operator priority

Highest priority at the top:

| | |
|---|---|
| `( )` | function calls and parentheses |
| `'` | postponed, - unary minus, ! - logical negation |
| `^` | power |
| `/, *` | multiplication operators |
| `+, -` | additive operators |
| `<, <=, ==, =>, >, !=` | comparison operators |
| `&` | logical and |
| `#, \|` | logical xor, or |
| `if then else` | conditional operator |

| | |
|---|---|
| = | assignment, function definition |

Two or more operators with equal priority are evaluated from left to right: `a+b-c = ((a+b)-c).`

""